Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 90 (2003) 45–47
www.elsevier.com/locate/entcs

Iterate, Incrementalize, and Implement: A Systematic Approach to Efficiency Improvement and Guarantees

Yanhong Annie Liu

Computer Science Department, SUNY Stony Brook, USA
liu@cs.sunysb.edu

People may more easily specify what they would like to compute if not concerned with how to compute efficiently. Given descriptions that may lead to inefficient computations, can we systematically obtain efficient algorithms and implementations? Can we further have efficiency guarantees for them?

We give an overview of a general and systematic method for achieving efficiency improvement and providing efficiency guarantees. The method is centered around incrementalization: making computation proceed repeatedly on small incremented inputs and effectively maintaining and using previously computed results. The method has three steps: iterate, incrementalize, and implement. Step 1 determines a minimum step to be taken repeatedly. Step 2 makes each step compute incrementally, particularly by using and maintaining and appropriate additional values. Step 3 designs appropriate data structures for storing and accessing the values maintained.

The central step of incrementalization [4] is: given a program f and an increment operation +, derive an incremental program that computes f(x+y) efficiently by using the result of f(x) [13], the intermediate results of f(x) [11], and auxiliary information of f(x) that can be inexpensively maintained [12]. This unifies existing approaches to incremental computation and is generally applicable; it exploits many existing program analysis and transformation techniques and can be systematically applied.

The method has been used successfully in optimizing expensive recursive

doi:10.1016/S1571-0661(03)00007-0

functions and recursive data structures [11,12,8,7,10], optimizing loops and aggregate array computations [3,5], transforming recursion into iteration [6], implementing sets and fixed points by Paige et al. [14,2,1], and currently implementing rules [9] and optimizing objects. Example applications include problems in list processing, graph algorithms, VLSI design, image processing, program analysis, and database queries.

These optimizations yield drastic algorithmic improvements. For example, incrementalizing recursive equations allows dynamic programming programs to be automatically derived [8,7]. Not only are the resulting programs drastically faster, but the time and space complexities of the resulting programs can also be much more easily calculated. In particular, in the most recent work on implementing Datalog rules [9], the time and space complexities of the resulting algorithms can be calculated from the rules. A prototype system, CACHET, for optimizations based on incrementalization has been implemented and gradually extended.

References

[1] J. Cai, P. Facon, F. Henglein, R. Paige, and E. Schonberg. Type analysis and data structure selection. In B. Möller, editor, *Constructing Programs from Specifications*, pages 126–164. North-Holland, Amsterdam, 1991.

[2] J. Cai and R. Paige. Program derivation by fixed point computation. *Sci. Comput. Program.*, 11:197–261, Sept. 1988/89.

[3] Y. A. Liu. Principled strength reduction. In R. Bird and L. Meertens, editors, *Algorithmic Languages and Calculi*, pages 357–381. Chapman & Hall, London, U.K., 1997.

[4] Y. A. Liu. Efficiency by incrementalization: An introduction. *Higher-Order and Symbolic Computation*, 13(4):289–313, Dec. 2000.

[5] Y. A. Liu and S. D. Stoller. Loop optimization for aggregate array computations. In *Proceedings of the IEEE 1998 International Conference on Computer Languages*, pages 262–271. IEEE CS Press, Los Alamitos, Calif., 1998.

[6] Y. A. Liu and S. D. Stoller. From recursion to iteration: what are the optimizations? In *Proceedings of the ACM SIGPLAN 2000 Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 73–82. ACM, New York, 2000.

[7] Y. A. Liu and S. D. Stoller. Program optimization using indexed and recursive data structures. In *Proceedings of the ACM SIGPLAN 2002 Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 108–118. ACM, New York, 2002.

[8] Y. A. Liu and S. D. Stoller. Dynamic programming via static incrementalization. *Higher-Order and Symbolic Computation*, 16(1-2):37–62, Mar.-June 2003. An earlier version appeared in *Proceedings of the 8th European Symposium on Programming*, 1999.

[9] Y. A. Liu and S. D. Stoller. From Datalog rules to efficient programs with time and space guarantees. In *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, Uppsala, Sweden, Aug. 2003.

[10] Y. A. Liu and S. D. Stoller. Optimizing Ackermann's function by incrementalization. In *Proceedings of the ACM SIGPLAN 2003 Workshop on Partial Evaluation and Semantics-Based Program Manipulation*. ACM, New York, 2003.

[11] Y. A. Liu, S. D. Stoller, and T. Teitelbaum. Static caching for incremental computation. *ACM Trans. Program. Lang. Syst.*, 20(3):546–585, May 1998. An earlier version appeared in *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 1995.

[12] Y. A. Liu, S. D. Stoller, and T. Teitelbaum. Strengthening invariants for efficient computation. *Sci. Comput. Program.*, 41(2):139–172, Oct. 2001. An earlier version appeared in *Conference Record of the 23rd Annual ACM Symposium on Principles of Programming Languages*, 1996.

[13] Y. A. Liu and T. Teitelbaum. Systematic derivation of incremental programs. *Sci. Comput. Program.*, 24(1):1–39, Feb. 1995.

[14] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Trans. Program. Lang. Syst.*, 4(3):402–454, July 1982.